A Hybrid Constraint Representation and Reasoning Framework

Keith Golden[1] and Wanlin Pang[2]

[1] Computational Science Division, NASA Ames Research Center, Moffett Field, CA 94035
[2] QSS Group Inc., NASA Ames Research Center, Moffett Field, CA 94035
{kgolden | wpang}@email.arc.nasa.gov

Abstract. In this paper, we introduce JNET, a novel constraint representation and reasoning framework that supports procedural constraints and constraint attachments, providing a flexible way of integrating the constraint system with a runtime software environment and improving its applicability. We describe how JNET is applied to a real-world problem – NASA's Earth-science data processing domain, and demonstrate how JNET can be extended, without any knowledge of how it is implemented, to meet the growing demands of real-world applications.

1 Introduction

Constraint-based reasoning has been shown to be useful in representing and reasoning about such diverse problems as *graph coloring*, *scene labeling*, *resource allocation* [27,24], and *planning and scheduling* [10,11,21]. In theory, the problem in hand is formalized as a constraint satisfaction problem (CSP) and is solved by using CSP algorithms such as backtracking. In practice, real-world applications often involve complex constraints that do not fall into the constraint definition in textbooks or in research papers, and are hard or impossible to model with built-in constraints in commercial constraint systems [1,2,6]. It is indeed the case that more and more commercial systems are available with more comprehensive built-in constraint libraries, and some even allow users to extend the constraint library by implementing domain specific constraints. However, extending a constraint library is not a ultimate solution [26], not mention being a great burden for the user of a constraint system to extend the system itself. There are many real-world applications, for example, the application of constraint-based planning to processing earth-observing satellite data [13], where the constraints involved are arbitrarily complex and dynamic, extending a constraint library may not be sufficient or even feasible.

We are applying constraint-based planning to software domains like the Earth-science data processing domain. Earth-science data processing is the problem of transforming low-level observations of the Earth system, such as data from Earth-observing satellites and ground weather stations, into high-level observations or predictions, such as *crop failure* or *high fire risk*. Given the large number of socially and economically important variables that can be derived from the data, the complexity of the data processing needed to derive them and the many terabytes of data that must be processed each day, there are great challenges and opportunities in processing the data in a timely manner, and a need for more effective automation. Our approach to providing this automation is to cast it as a planing problem: we represent data-processing operations as

planner actions and desired data products as planner goals, and use a planner to generate data-flow programs that produce the requested data.

Constraints arise naturally in this planning problem. Specifications of data inputs and outputs include constraints indicating geographic regions of interest, thresholds on resolution, data quality, file size, etc. Specifications of data-processing operations include constraints relating the inputs of the operations to the outputs, where inputs and outputs are complex objects such as satellite images and weather forecast data. For example, scaling an image creates a new image whose dimensions are some multiple of the dimensions of the original. In the course of planning, additional constraints arise, specifying how parameters of an action depend on the parameters of other actions in the plan. Because of the complex objects involved in the constraints and because the world is large and dynamic, it is impossible to enumerate in advance all possible objects, such as satellite images, much less provide an explicit representation of the constraints among them. Most significantly, many of the constraints in the data processing domain are very complex, but are implemented as executable code in a software environment. Reimplementing them in a constraint reasoning system would not only be difficult, but would also violate the principle that information should exist in only one place.

To make matters worse, even the *planning domain* can change dynamically. In order to accommodate the changing availability of data feeds and other resources and to incorporate new Earth system models, some of which are produced automatically by machine learning algorithms, the planner must be able to handle the addition and removal of planner actions, data types and other components in a *plug-and-play* fashion, even during plan construction. Some of these definitions, such as those corresponding to newly learned Earth system models, are previously unseen components received remotely by the planner. Since any of these definitions can include arbitrary constraint definitions, the set of constraints that are available also changes dynamically.

We would like to integrate the constraint reasoning system with the runtime software environment so that the operations provided by the environment can be used as constraints. We would like the constraint system to *query* the environment, to dynamically determine what objects exist and what attributes or properties those objects have. Doing so requires being able to define types in the constraint system that correspond to entities within the runtime environment and to define constraints in terms of operations supported by the runtime environment.

The procedural constraint reasoning framework (we will call it CNET, for Constraint NETwork) introduced in [18,19], comes close to providing the capabilities we need. CNET allows implementation of procedure constraints, but in a static Constraint library. It would be difficult, if not impossible, to code the complex and dynamic constraints in the data processing domain.

We have implemented a hybrid constraint reasoning system, called JNET (for Java constraint NETwork), as a component of the planner-based agent called IMAGEbot [13], as shown in Figure 1. JNET builds upon the CNET, not only extending the CNET constraint library, but more importantly, providing a better way to model domain specific constraints; that is, it allows arbitrary, complex constraints to be defined at runtime in a plug-and-play fashion using *constraint attachments*, constraints specified in terms of functional Java methods, which are concise and simple to specify without any

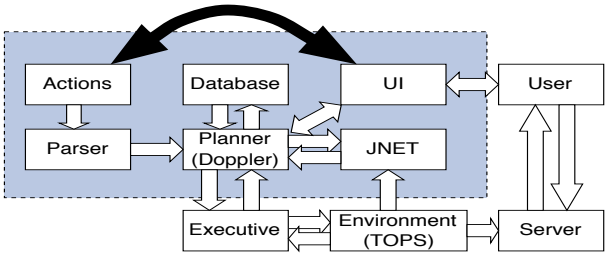

Fig. 1. The architecture of IMAGEbot

knowledge of the workings of JNET but which interact well with the JNET search and propagation algorithms.

The contributions of JNET include the use of regular expressions to represent string domains [15], support for universally quantified constraints [14], constraints over structured objects, and planning-graph-based constraint propagation algorithms (to be reported), but this paper focuses on the *constraint attachments* which allow JNET to directly interact with a runtime environment, greatly improving applicability of constraint techniques.

The remainder of the paper is organized as follows: In Section 2, we present the JNET framework. In Section 3, we discuss the application of JNET to data processing. In Section 4, we briefly review related work. In Section 5, we summarize our contribution.

2 Procedural Reasoning Framework

The architecture of IMAGEbot is described in Figure 1. A planning problem, specified in the DPADL language [12], is reformulated in the IMAGEbot Planner as a CSP. This CSP is handled by JNET, which will be discussed in this section. We begin by reviewing some needed CSP concepts and notations.

2.1 Constraint Satisfaction Problems

A Constraint Satisfaction Problem (CSP) consists of variables, domains, and constraints. Formally, it can be defined as a triple $<X,D,C>$ where $X = \{x_1, x_2, \ldots, x_n\}$ is a finite set of variables, $D = \{d(x_1), d(x_2), \ldots, d(x_n)\}$ is a set of domains containing values the variables may take, and $C = \{C_1, C_2, \ldots, C_m\}$ is a set of constraints. Each constraint C_i is defined as a relation R on a subset of variables $V = \{x_i, x_j, \ldots, x_k\}$, called the constraint scope. R may be represented extensionally as a subset of the Cartesian product $d(x_i) \times d(x_j) \times \ldots \times d(x_k)$. A constraint $C_i = (V_i, R_i)$ limits the values the variables in V can take simultaneously to those assignments that satisfy R. A solution to the problem is an assignment of values to variables in X satisfying constraints in C. The central reasoning task (or the task of solving a CSP) is to find one or more solutions.

Many search and propagation algorithms have been developed for solving constraint problems [22]. However, constraints involved in real-world applications, such as the data-processing domain, represent new challenges as to how to represent these constraints. Constraint procedures are introduced to address the issue.

2.2 Constraint Procedures

The idea of procedural reasoning in constraint satisfaction [18] is to augment a general constraint search algorithms with specific procedural methods that can quickly solve certain types of subproblems and prune a search space that contains no potential solutions. In a certain sense, similar techniques have been widely used in solving binary CSPs: that is, enforcing arc-consistency while searching for solutions by backtracking [25]. There have been many algorithms published for enforcing arc-consistency [3], but the question of how to detect and then remove inconsistent values has largely been ignored, because it seems to be a trivial implementation issue when dealing with binary constraints, which can be uniformly represented as a 0-1 matrix (assuming finite domains). However, when it comes to non-binary constraints, enforcing local consistency relative to a constraint is not obviously a trivial task. For example, let $z > x + y$ be a constraint on 3 integer variables over domain $\{0,1,2\}$. We know that z cannot take 0, that x and y cannot take 2 simultaneously, and that $< 1,0,1 >$ is not a consistent triple, but how to identify these inconsistent values, inconsistent value pairs and triples is not trial anymore. Even for this simple constraint, there are many ways to implement it; some may eliminate inconsistent values, and some may eliminate value pairs or triples.

In fact, constraints in different application domains are represented and enforced in different ways. In addition, many constraint problems contain simple functional relations (e.g. arithmetic equations) and simple subproblems (e.g. linear equations with unknowns) that can be solved quickly by using existing algorithms. The question becomes 1) how to uniformly represent constraints that arise in different applications; 2) how to take advantage of such algorithms in order to significantly improve search efficiency. The procedural reasoning framework addresses this question [18].

A constraint procedure p is a function that maps a CSP $\mathbb{P} = (X, D, C)$ to another CSP $\mathbb{P}' = (X, D', C')$ such that: 1) $d'(x_i) \subseteq d(x_i)$ for each $x_i \in X$, $d(x_i) \in D$, $d'(x_i) \in D'$; 2) for each constraint $C_h = (V_h, R_h) \in C$ there exists a constraint $C'_h = (V_h, R'_h) \in C'$, such that C_h and C'_h have the same scope and $R'_h \subseteq R_h$.

The concept of constraint procedures provides a uniform and efficient method to represent and reason with constraints. In terms of representation, constraint procedures can be used to define any kind of constraints over any kind of variables. In fact, by the CSP definition in Section 2.1, a constraint on a subset of variables can be seen as a function that maps the universal relation on the variable subset into a restricted relation on the same variables defined by the constraint. Take constraint $z > x + y$ as an example; the universal relation on these 3 variables is a set of 27 triples $\{< 0,0,0 >, < 0,0,1 > ..., < 2,2,2 >\}$, and the restricted relation on these 3 variables is a set of value triples without any inconsistent ones. If the constraint can be seen as a function mapping this universal relation to the restricted relation, then it can be represented as a procedure, which is suppose to eliminate inconsistencies when executed.

From the reasoning point of view, constraint procedures can be applied for the purpose of both maintaining consistency and searching for a solution. For example, if $< 1,0,1 >$is assigned to $< z,x,y >$, executing the constraint procedure should return a failure. However, it may do more than just maintaining consistency. For example, when 1 is assigned to z, a procedure can eliminate 2 from the domains of x and y. If $< 1,0 >$ is assigned to $< z,x >$, the procedure can eliminate both 1 and 2 from the

domain of y. Now variable y has only one value left, so there is no need for search on this variable.

2.3 Constraint Attachments

The idea of constraint procedures is great, but how to implement constraint procedures is another issue. The good news is that the procedural reasoning framework – CNET – is implemented in C++ at NASA Ames and it has been successfully applied to solving constraint problems in many NASA missions (e.g., the MAPGEN Planner, used to generate command sequences for the Mars Exploration Rovers, uses part of it) . However, it still falls short of our requirements in the data processing domain, as discussed in Section 1, which leads us to extend the framework to include *constraint attachments*.

A key goal of constraint attachments is to allow domain-specific constraints to be expressed intuitively, with no knowledge of the internals of the constraint reasoning system (e.g., JNET in this case), and loaded or unloaded at runtime in a plug-and-play fashion. Thus, the constraints should contain no reference to JNET API calls or data structures such as Variable, Value, Domain, etc.

In keeping with our goal of simplicity, we define a constraint attachment as a set of functional methods, each of which determines the value(s) of one variable based on the value(s) of the others. This decision is based entirely on the fact that functions and variable assignment are such a familiar idioms, being the building blocks of the most popular programming languages. Each method takes a list of arguments as input variables and returns the calculated result for its output variable. For example, the constraint $x+y=z$ would, in general, include three methods: $z \leftarrow x+y$, $x \leftarrow z-y$, and $y \leftarrow z-x$. The method $z \leftarrow x+y$ calculates z's domain based on the domains of the given variables x and y, and it is usually invoked when either or both domains of x and y changes.

Formally, an attachment is a pair <P,m>, where P is a *signature* and m is a *method*. Conceptually, P specifies the arguments and return values of the method m as a list, $\{a_0, a_1, \ldots a_n\}$, where the first argument, a_0, designates the variable that will be assigned the return value of m and $a_1, \ldots, a_n$ designate the variables that will provide the arguments to m. The arguments a_i are not just variables, however. If we were only interested in implementing attachments that took singletons as arguments and returned singletons as results, then all we would need for P would be a list of variables. Instead, we allow the domain modeler to specify that an argument represents an entire domain, which may be in the form of a finite set or an interval. Thus, in addition to the variable, it is also necessary to specify what form the domain should take: a singleton, set or interval. Each argument a_i, then, is a pair <x_i,t_i>, where x_i is a variable from the constraint network and t_i specifies the form that the domain of x_i should take; $t_i \in \{1, \lfloor \Im \rfloor, \lceil \Im \rceil, \mathcal{S}\}$, where 1 is used to denote a singleton, $\lfloor \Im \rfloor$ and $\lceil \Im \rceil$ denote the lower and upper bound of an interval, respectively, and $\mathcal{S}$ denotes a finite set. The method m will only be applicable if each of the domains $d(x_i)$ can be converted to the representation t_i required by m. For example, if $t_i = \lfloor \Im \rfloor$, then $d(x_i)$ must be a numeric domain whose values all fall within a given closed interval, with no gaps. If that is the case, then the value of $a_i$will be the lower bound of that interval. The choice of domain representation is based on what can conveniently be expressed without reference to JNET internals or special data structures. In particular, by splitting the calculation of interval domains into

separate calculations on the upper and lower bounds, we ensure that the user-supplied code need only refer to single values of the appropriate type or sets, both of which are provided by all major programming languages.

2.4 Comparing attachments to procedures

In general, the notion of a constraint procedure encompasses a wide range of constraint reasoning techniques, from simple propagation to complete search methods. For example, any constraint algorithm that finds all solutions to a given CSP can be considered as a procedure constraint over all variables, because it is a function that maps the universal relation on all variables to a restricted relation containing all solutions. Constraint attachment, as a different constraint representation techniques, can also be considered a special case of constraint procedure.

An advantage of constraint attachment over constraint procedure is its flexibility in representing and reasoning with constraints. Given a constraint attachment, not only can a set of functional methods be selectively implemented, but also the implemented methods can be selectively executed by a constraint propagator or a constraint solver without any tailoring of the propagator and the solver to the specific constraints. This selective execution can exploit the knowledge of what variable domain a given method will affect and what variable domains that method depends on, using the signature *P*. The more general constraint procedures are more opaque.

Most importantly, constraint attachment allows constraint to be defined dynamically at runtime and allows executables in the application domains to be invoked by constraint execution, which provides a flexible way of integrating the constraint system with a runtime software environment and significantly improves applicability of constraint systems. We will discuss in detail how JNET interacts with a dynamic runtime environment in Section 3.

2.5 Implementation

JNET is implemented in Java. It contains classes for variables, domains, constraints, and search and propagation algorithms. Each variable is associated with a domain. A variable domain can be finite or infinite, in which case it is represented as an an interval (for numeric types), regular expression (for string types), or symbolic set (for object types).

The JNET constraint solver contains several search algorithms, including depth-first search, backjumping and conflict-directed backjumping, all interleaved with the JNET propagator, which controls the execution of constraints; that is, when the solver assigns a value to a variable, the propagator will execute all those constraints containing the variable and will continue to execute constraints until there are no more changes. The propagation essentially maintains *generalized arc-consistency* [20]. If a variable domain becomes a singleton during propagation, it is considered to have a value assignment and is removed from the searchable variable set to avoid unnecessary search. Therefore, propagation plays an important role in the problem solving process.

The constraints are implemented as procedures or constraint attachments. A procedural constraint consists of a set of variables (the scope) and a procedure (i.e., an

Algorithm 1 Implementation of Constraint $x+y<z$

Let $d(x)=[xmin,xmax]$, $d(y)=[ymin,ymax]$, and $d(z)=[zmin,zmax]$

execute($X=\{x,y,z\}$)

1. **if** $d(x)$, $d(y)$, or $d(z)$ is empty, **return failure**;
2. $zmin \leftarrow xmin+ymin$; $xmax \leftarrow zmax-ymin$; $ymax \leftarrow zmax-xmin$;
3. **if** (zmin $\neq -\infty$) $d(z) \leftarrow d(z) \cap [zmin,zmax]$
4. **if** $d(z)$ is empty **return failure**;
5. **if** (xmax $\neq \infty$) $d(x) \leftarrow d(x) \cap [xmin,xmax]$
6. **if** $d(x)$ is empty **return failure**;
7. **if** (ymax $\neq \infty$) $d(y) \leftarrow d(y) \cap [ymin,ymax]$
8. **if** $d(y)$ is empty **return failure**;
9. **return success**;

Algorithm 2 Implementation of the attach constraint

Let P be the signature and m the Java method as defined in Section 2.3.

`execute(`<P,m>`)`

1. **`for each`** $< x_i,t_i > \in P$ `where` $i>0$
   (a) **`if`** ($d(x_i)$ `is not representable as a domain of type` $t_i \in \{1, \lfloor \Im \rfloor, \lceil \Im \rceil, \mathcal{S}\}$) **`return success;`**
   (b) **`else let`** d_i `be` $d(x_i)$ `represented as type` t_i**`;`**
2. **`let`** $v_0 \leftarrow$ **`invoke`** $m(d_1,\ldots,d_n)$
3. **`assign`** $d(x_0) \leftarrow d(x_0) \cap \begin{cases} \{v_0\} & \text{if } t_0=1 \\ v_0 & \text{if } t_0=\mathcal{S} \\ [v_0,\infty) & \text{if } t_0=\lfloor \Im \rfloor \\ (-\infty,v_0] & \text{if } t_0=\lceil \Im \rceil \end{cases}$
4. **`if`** ($d(x_0)$ is empty) **`return failure`**
5. **`return success;`**

`execute()` method) that enforces the underlying constraints on the variables. Executing a constraint eliminates inconsistent values from the domains of variables in the scope; that is, it examines the current domains and eliminates any values that are not consistent with values remaining in domains of other constrained variables. If execution of a constraint results in an empty variable domain, `execute()` returns failure indicating the violation of the constraint. In the current version, the JNET constraint library contains about 30 application-independent constraints, such as `equality`, `less-than`, `maximum` and `minimum`, `cardinality`, regular expression `match` and string `concatenation`. For example, an implementation of constraint $x+y<z$, where x, y, and z are interval variables, is described in Algorithm 1.

Constraint attachments are implemented on top of procedure constraints. In particular, there is a procedure constraint, `attach`, in JNET, which implements the `execute(`<P,m>`)` method for an attachment <P,m>, as shown in Algorithm 2.

Constraint attachment allows the JNET users to define domain-specific constraints not included in the JNET library by implementing some attached functional methods and creating new `attach` constraint instances to call these methods. More conveniently, new instances of `attach` constraints can be created to call the methods or algorithms already implemented in the run-time environment to enforce the user constraints, which we discuss in the next section.

3 TOPS Case Study

As a demonstration of our approach, we have applied our constraint-based planning system—IMAGEbot—to the Terrestrial Observation and Prediction System (TOPS, http://www.forestry.umt.edu/ntsg/Projects/TOPS/), an ecological forecasting system that assimilates data from Earth-orbiting satellites and ground weather stations to model and forecast conditions on the surface, such as soil moisture, vegetation growth and plant stress. Prospective customers of TOPS include scientists, farmers and fire fighters. With such a variety of customers and data sources, there is a strong need for a flexible mechanism for producing the desired data products for the customers, taking into account the information needs of the customer, data availability, deadlines, resource usage (some scientific models take many hours to execute) and constraints based on context (a scientist with a palmtop computer in the field has different display requirements than when sitting at a desk). IMAGEbot provides such a mechanism, accepting goals in the form of descriptions of the desired data products.

The goal of the TOPS system is the monitoring and prediction of changes in key environmental variables. The inputs needed by TOPS include satellite data, such as Fractional Photosynthetically Active Radiation (FPAR), and weather data, such as precipitation. There are several potential candidate data sources for each input required by TOPS at the beginning of each model run. The basic properties of the inputs are listed in Table 1. Even with this fairly small model, there is a good variety of inputs we need

Source	Variables	Frequency	Resolution	Coverage
Terra-MODIS	FPAR/LAI	1 day	1km, 500m, 250m	global
Aqua-MODIS	FPAR/LAI	1 day	1km, 500m, 250m	global
AVHRR	FPAR/LAI	10 day	1km	global
SeaWIFS	FPAR/LAI	1 day	1km x 4km	global
DAO	temp, precip, rad, humidity	1 day	1.25 deg x 1.0 deg	global
RUC2	temp, precip, rad, humidity	1 hour	40 km	USA
CPC	temp, precip	1 day	point data	USA
Snotel	temp, precip	1 day	point data	USA
GCIP	radiation	1 day	1/2 deg	Continental
NEXRAD	precipitation	1 day	4 km	USA

Table 1. TOPS input data choices

to select from, depending on the desired data products.

In addition to the attributes listed in the table, data sources also vary in terms of quality and availability—some inputs are not always available even though they should be. For example, both the Terra and Aqua satellites experienced technical difficulties and data dropouts over periods ranging from few hours to several weeks. Depending on the data source, different processing steps will be needed to get the data into a common format. We have to convert the point data (CPC and Snotel) to grid data, which by itself is a fairly complex and time-consuming process, and we must reproject grid data into a common projection, subset the dataset from its original spatial extent, and populate the input grid used by the model. The data are then run through the TOPS model, which generates the desired outputs.

The architecture of IMAGEbot is described in Figure 1. Planning domains are specified in a language called the Data Processing Action Description Language (DPADL) [12], which allows the description of planning domains that involve data processing operations as well as the constraints appearing in those domains.

The planning specification, which contains object types and attributes, functions and relations, as well as planner actions and goals, is loaded by the parser and passed to the planner. The planner constructs a data structure we call a *lifted planning graph* to concisely represent the search space. A lifted planning graph is similar to the planning graph data structure used by Graphplan-based planners [4]. A planning graph is a leveled graph of alternating "proposition" and "action" levels, in which the first level nodes comprise all propositions that are true in the initial state, the second level nodes represent all ground actions whose propositions are present in the first level, the third level contains all propositions achievable by executing actions in the second level, and so on. Arcs are present between an action and the propositions in its preconditions (prior level) and its postconditions (following level). Because a grounded representation is not an option in data processing domains, where the universe is large, uncertain and dynamic, we use a *lifted* planning graph, which can contain variables and constraints among those variables, and in which the "proposition" level mainly comprises objects, with arcs to the inputs and outputs of actions. For example, the planning graph in 2, constructed from a planning specification file in Figure 4, contains objects, shown as round nodes, and actions shown as rectangular nodes.

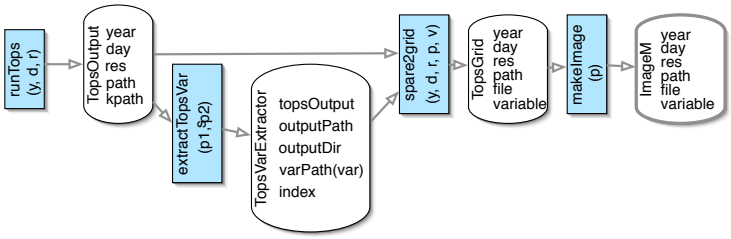

Fig. 2. A portion of the planning graph corresponding to a plan from the TOPS domain. The rectangular nodes represent actions and the round nodes represent objects that are input/output by the actions. The labels inside object nodes are attributes of the objects.

After the initial graph is constructed, a CSP representing the search space is built. The uncertain and dynamic nature of the planning problem requires us to interleave

planning, constraint reasoning and execution, so in general there are subsequent iterations, in which the graph is expanded, dead ends are pruned, and the corresponding variables and constraints are added or removed from the CSP. This requires JNET to be able to handle dynamic CSPs.

The CSP corresponding to the planning graph in Figure 2 contains 776 variables and 964 constraints. There are 280 boolean variables representing the arcs (causal links) and conditions in the plan, 164 integer variables and 111 string variables representing action parameters and object attributes, and 107 object variables representing TOPS objects and instances, as well as action parameters and object attributes. Except for boolean variables, most variables initially have infinite domains, represented as symbolic sets, intervals, and regular expressions for object, integer, and string variables respectively.

There are 326 instances of `attach` constraints for interaction with the TOPS environment, and the rest are procedural constraints reflecting the planning problem. A part of the constraint graph from the planing graph in Figure 2 is shown in Figure 3. The constraints, for example, CondEqual(*`tout.year=y`*, *`tout.year`*, *`y`*), means that

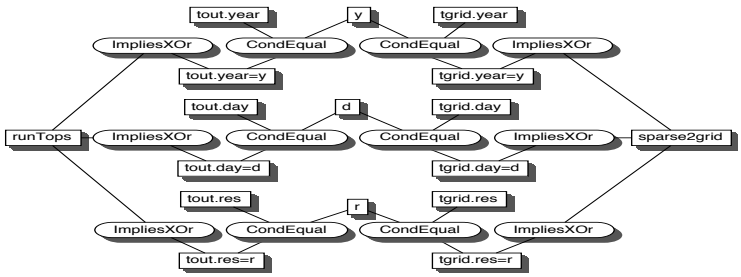

Fig. 3. A portion of the constraint network from the planning graph in Figure 2.

variable *`tout.year=y`* is true iff variable *`tout.year`* equals variable *`y`*. In general, we have boolean constraints such as

ImpliesXOR($c, x_1, \ldots, x_n$), *i.e.*, $c \Rightarrow x_1 \otimes \ldots \otimes x_n$
CondOr($c, x_1, \ldots, x_n$), *i.e.*, $c \Leftrightarrow x_1 \vee \ldots \vee x_n$
CondAnd($c, x_1, \ldots, x_n$), *i.e.*, $c \Leftrightarrow x_1 \wedge \ldots \wedge x_n$

numerical constraints such as

Min($z, x_1, \ldots, x_n$), *i.e.*, $z = min\{x_1, \ldots, x_n\}$
Product($z, x_1, \ldots, x_n$), *i.e.*, $z = x_1 \times \ldots \times x_n$

string constraints such as

Match(c, x, y), *i.e.*, $c \Leftrightarrow match(x, y)$
Concat(z, x, y), *i.e.*, $z = x + y$

and others, all predefined in the JNET constraint library.

As mentioned in Section 2.5, there is only one `attach` constraint class implemented in the JNET library; the user implements attached methods and creates new instances of `attach` constraints to call the attached methods. In IMAGEbot, however, the parser

generates attachments from the DPADL specification by generating Java methods from in-lined code (delimited by $...$) appearing in attachment definitions, together with the signature, from which the method's parameter list and return type can be determined. The parser then compiles and loads the generated Java code, and uses reflection to obtain references to the newly defined methods from their names.

For example, the following segment of DPADL code specifies that type `Tile` corresponds to the Java class `tops.modis.Tile`, which is a rectangular satellite image in some specified projection covering some definite region of the Earth. A tile has a number of attributes, including the projection, the instrument used to capture the image, the time and location where the image was captured, the pathname where the image is stored and a unique identifier that can be used to reference the tile.

```
type Tile isa object mapsto tops.modis.Tile {
  ......
  key String uniqueId {
    constraint {
      value(this) := $this.getUId()$;
      this(value) = $Tile.findTile(value)$;
    }
  }

  boolean covers(float lon, float lat) {
    constraint {
      ......
      {this}([lon],[lat],d, y, p, value) = {$ if (value)
      return tm.getTiles(lon.max, lat.min, lon.min, lat.max, d, y, p);
      else return null; $};
    }
  }
  ......
}
```

The attribute `uniqueId` is declared as a **`key`** of a tile, meaning there must be a *one-to-one* mapping between tiles and their unique identifiers. The constraints that enforce this one-to-one correspondence are specified using an `attach` constraint instance with two methods, generated by the parser:

<{<x_{value}, 1>, <x_{this}, 1>}, String m_1(Tile x){**return** x.getUID();}>
<{<x_{this}, 1>, <x_{value}, 1>}, Tile m_2(String x){**return** Tile.findTile(x);}>

where x_{value} and x_{this} are variables, the value 1 is the singleton domain type as discussed in Section 2.3.

The value 1 in the above attachments means that variable x_{value} can only be assigned a value if variable x_{this} is singleton, and vice versa. It is also possible to define constraints that work for non-singleton domains, by indicating that an argument or return value represents the upper or lower bound of an interval ($\lfloor \Im \rfloor$, $\lceil \Im \rceil$) or a finite set $\mathcal{S}$. For example, another attribute of a `Tile` in the above DPADL code is that it `covers` a given longitude and latitude. Given a particular longitude and latitude, the constraint solver can invoke a method to find a single tile that covers it, but it can do even better.

Given a rectangular region, represented by intervals of longitude and latitude, it can invoke a method to find a set of tiles covering that region. In this case, $P = \{<x_{this}, \mathcal{S}>, <x_{lon}, \lfloor \Im \rfloor>, <x_{lon}, \lceil \Im \rceil>, <x_{lat}, \lfloor \Im \rfloor>, <x_{lat}, \lceil \Im \rceil>, <x_d, 1>, <x_y, 1>, <x_p, 1>, <x_{value}, 1>\}$ and m is a method that returns a Collection of `Tile` objects.

The above methods, generated from the inline code and the method signature, are compiled and loaded when the type definition is parsed, and references to the methods are obtained using Java reflection. This enables constraint attachments to be defined at runtime, without requiring modification to, or even access to, the source code of JNET or constraint library.

Once created and added to JNET, both types of constraints—procedures and attachments—are treated uniformly: whenever the domain of a variable changes, the constraints associated with the variable are executed to propagate the changes, which is controlled by JNET propagators.

The planner controls the high-level search, guided by heuristics derived from a Graphplan-style [4] reachability analysis. JNET ensures the underlying CSP is consistent by propagating changes made by the planner. After a plan is generated, JNET does its own search to find values to variables representing action parameters to make actions executable. This is an iterative process involving possible backtracks; that is, if there are no valid parameters for a chosen action, the planner has to search for another plan; if it is impossible to extract a plan from the current planning graph, the planning graph has to be extended. At the end, we have a plan and a data product resulting from executing the plan, as in Figure 4.

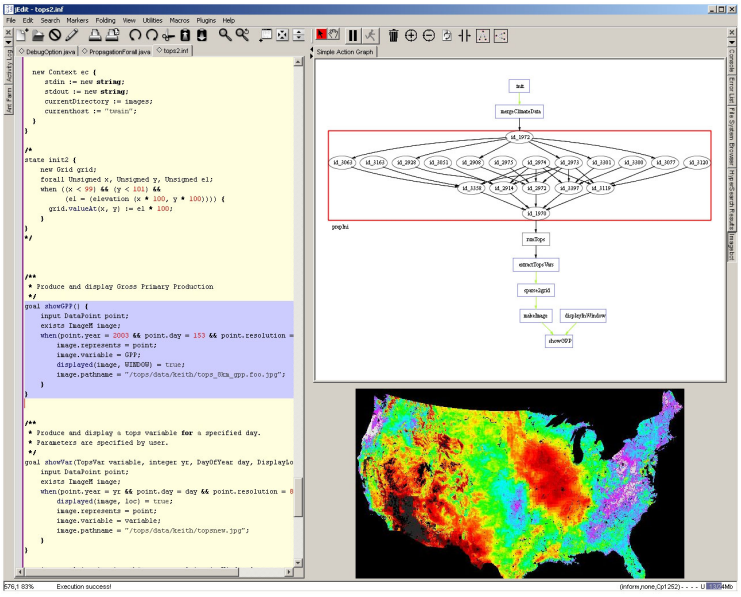

Fig. 4. The IMAGEbot development environment, running as a jEdit plugin. The frame on the left shows one of the files comprising the TOPS domain description. The frame on the upper right shows an abstracted view of a plan from Figure 2, with one action node displayed in detail. The frame on the lower right is the data returned by the TOPS server after executing the plan.

4 Related Work

There have been a good number of constraint systems developed in recent years. Many have been successfully applied to various real-world combinatorial optimization problems such as planning and scheduling, resource allocation, time-tabling, configuration, etc., and more will be developed in order to meet growing application needs in terms of enhanced modeling and solving capacity.

Early systems such as CHIP [2], CLP(R) [17], and Prolog III [9] extend Prolog with additional constraint solving over a particular domain of interest, where the constraint solver in the system works as a *black box* responsible for constraint propagation. The black-box approach, which has very limited modeling power, has evolved into *glass-box* approach, as in CLP(FD) [8], Eclipse [6], ILOG Solver (http://www.ilog.-com), JCL (http://liawww.epfl.ch/~torrens/Project/JCL/), and Koalog Constraint Solver (http://www.koalog.com/), where the underlying constraint solver can be tailored to the users' needs, and user constraints can be defined and added to the constraint system. For example, CLP(FD) is a constraint logic programming language with finite domain constraints. Its implementation is based on the use of a single primitive constraint, variable X in range r, that embeds the core propagation techniques such as node and arc consistencies. The more complex user constraints such as linear equations or inequations can be defined and compiled into the primitive constraint. Constraint programming tools, ILOG Solver (and JSolver), and Koalog Constraint Solver (which was first released in later 2002), packaged a set of primitive constraints and of implemented basic constraint search algorithms. These tools are open and extensible in that new constraints can be added, and new search algorithms and heuristics can be developed by the user.

Compared to the constraint systems mentioned above, the constraint procedure framework CNET [19] shares similar theoretical principles, except the constraints are represented uniformly in a more general form—procedure. The procedural reasoning framework (CNET), which was developed in *C++* as at NASA Ames, has many capabilities that are missing from previous constraint systems, such as: i) arbitrary constraints—there is no limit on the types of constraints that can be handled; ii) dynamic variables and domains—the set of variables and their values need not be enumerated beforehand; iii) real-valued variables—constraints may involve mixtures of discrete and continuous variables; and iv) hybrid reasoning—different reasoning techniques can be utilized within the same system.

JNET extends CNET to support *string domain*s, *quantifie*d, *structured objects*, and other features, including *constraint attachments*. The idea of constraint attachment can be traced back to *procedure attachment* in [16], where some specialized procedures (or functions) are implemented to evaluate certain variables and those procedures are *attached to solvers*. It has been noted in [18] that this kind of attachment applying to constraint satisfaction is very limited in terms of reusability, global algorithm implementation, and integration with constraint solvers. Our idea of attaching functional methods to the execution of constraints is motivated by the requirement of integrating JNET to TOPS (more specifically, the requirement of using executables in TOPS environment), and a general mechanism of how to do so is implemented in JNET. In addition to the advantages of CNET over other constraint reasoning tools, JNET provides the capability

of integrating constraint reasoning system with applications domains, which we believe is the right direction to improve the applicability of constraint technologies.

There has been little work in planner-based automation of data processing. Two notable exceptions are Collage [23] and MVP [7], both of which also rely heavily on constraints. Both of these planners were designed to provide assistance with data analysis tasks, in which a human was in the loop, directing the planner. Consequently, both planners are based on action decomposition, which is more intuitive to many users. In contrast, the data processing in TOPS must be entirely automated; there is simply too much data for human interaction to be practical. [5] addresses workflow planning for computation grids, a similar problem to ours, though their focus is on mapping pre-specified workflows onto a specific grid environment, whereas our focus is on generating the workflows.

5 Conclusion

We have described the JNET constraint reasoning system and discussed its application to a data-processing domain. JNET is implemented as a component of the IMAGEbot planner-based agent and it provides the planner with constraint reasoning capabilities. As a constraint reasoning system, JNET can be applied to solving constraint problems in other real-world application domains. To do so, the user needs to define variables and their domains, and specify the constraints using the predefined constraints in the constraint library. For modeling application-specific constraints that are not defined in the constraint library, JNET provides the user with two alternatives:

1. Constraints can be implemented as reusable procedural constraints by extending the constraint template provided in JNET. This is similar to how user constraints are defined in some other constraint systems.
2. Constraints can be implemented as a set of attached functional methods, which may be defined at runtime, without modification to, or even access to, the JNET source code. This is the feature makes JNET applicable to many real-world domains new to constraint technologies.

JNET provides an easy way to integrate non-constraint-based services into a constraint-based application; any Java classes can be used as types, and any methods provided by those classes can be used to implement constraints. This capability is used in IMAGEbot to integrate planning with sensing; *sensors* that return information about a software environment, such as the locations of files, are implemented as constraint attachments; as relevant variables become constrained, different sensors (in the form of attachments) are activated, yielding additional constraints which may, in turn, activate other sensors.

References

1. ILOG 2000. *ILOG Solver 5.0. User Manual*. ILOG, Gentilly, France, 2000.
2. N. Beldiceanu and E. Contejean. Introducing global constraints in CHIP. *Mathematical and Computer Modelling*, 20(12):97–123, 1994.

3. C. Bessiere and M. Cordier. Arc-consistency and arc-consistency again. In *Proceedings of AAAI-93*, pages 108–113, 1993.
4. A. Blum and M. Furst. Fast planning through planning graph analysis. *AIJ*, 90(1–2):281–300, 1997.
5. J. Blythe, E. Deelman, Y. Gil, C. Kesselman, A. Agarwal, G. Mehta, and K. Vahi. The role of planning in grid computing. In *Proc. 13th Intl. Conf. on Automated Planning and Scheduling (ICAPS)*, 2003.
6. A. Cheadle, W. Harvey, A.Sadker, J. Schimpf, K. Shen, and M. Wallace. ECLiPSe: an introduction. Technical report, Imperial College, London, UK, 2003.
7. S. Chien, F. Fisher, E. Lo, H. Mortensen, and R. Greeley. Using artificial intelligence planning to automate science data analysis for large image database. In *Proc. 1997 Conference on Knowledge Discovery and Data Mining*, August 1997.
8. P. Codognet and D. Diaz. Compiling constraints in clp(fd). *Journal of Logic Programming*, 27, 1996.
9. A. Colmerauer. An introduction to Prolog-III. *Communication of the ACM*, 33(7), 1990.
10. M. Drummond, J. Bresina, and K. Swanson. Just-In-Case scheduling. In *Proceedings of AAAI-94*, Seattle, WA, 1994.
11. J. Frank and E. Kurklu. SOFIA's choice: Scheduling observations for an airborne observatory. In *Proceedings of ICAPS-03*, Trento, Italy, 2003.
12. K. Golden. DPADL: An action language for data processing domains. In *Proceedings of the 3rd NASA Intl. Planning and Scheduling workshop*, pages 28–33, 2002. to appear.
13. K. Golden. Automating the processing of earth observation data. In *7th International Symposium on Artificial Intelligence, Robotics and Automation for Space*, 2003.
14. K. Golden and J. Frank. Universal quantification in a constraint-based planner. In *AIPS02*, 2002.
15. K. Golden and W. Pang. Constraint reasoning over strings. In *Proceedings of the 9th International Conference on the Principles and Practices of Constraint Programming*, 2003.
16. C. Green. *The application of theorem proving to question answering systems*. PhD thesis, Stanford University, 1969.
17. J. Jaffar, S. Michaylov, P. Stuckey, and R. Yap. The CLP($\mathcal{R}$) Language and System. *ACM Transactions on Programming Languages and Systems*, 14(3):339–395, July 1992.
18. A. Jónsson. *Procedural Reasoning in Constraint Satisfaction*. PhD thesis, Stanford University, 1996.
19. A. Jónsson and J. Frank. A framework for dynamic constraint reasoning using procedural constraints. In *Euopean Conference on Artificial Intelligence*, 2000.
20. G. Katsirelos and F. Bacchus. GAC on conjunctions of constraints. In *CP-2001*, 2001.
21. L. Khatib, J. Frank, R. Morris, D. Smith, and J. Dungan. Interleaved observation execution and rescheduling on Earth Observing Systems. In *Proceedings of ICAPS-03 Workshop on Plan Execution*, 2003.
22. G. Kondrak and P. van Beek. A theoretical evaluation of selected backtracking algorithms. *Artificial Intelligence*, 89:365–387, 1997.
23. A. L. Lansky and A. G. Philpot. AI-based planning for data analysis tasks. In *Proceedings of the Ninth IEEE Conference on Artificial Intelligence for Applications (CAIA-93)*, 1993.
24. N. Muscettola. Computing the envolope for stepwise constant resource allocations. In *Proceedings of CP-2002*, 2002.
25. B. A. Nadel. Consistent satisfaction algorithms. *Computational Intelligence*, 5:188–224, 1989.
26. M. Wallace. Languages versus packages for constraint problem solving. In *Proceedings of CP-2003*, Kinsale, Ireland, 2003.
27. M. Zweben and M. Fox. *Intelligent Scheduling*. Morgan Kaufmann Publishers, 1994.